THE RAND COMPUTATION CENTER:

WYLBUR COMMAND FACILITIES

Paul Andersen

David J. Smith

R-1555/10

May 1975

The Rand Computation Center
1700 Main Street Santa Monica, California 90406

PREFACE

This manual describes two related facilities that extend WYLBUR in the direction of a programming language that supports structured text entry and text manipulation applications.

The first facility is an extension to WYLBUR's command vocabulary that gives WYLBUR many of the capabilities associated with traditional programming languages. The extension was written by Paul Andersen.

The second facility is a batch preprocessor that permits the WYLBUR programmer to develop WYLBUR command programs in a language similar to PL/I. The preprocessor was developed by David J. Smith.

The authors wish to express their appreciation for the support given them by members of the Computation Center system staff. Eric Harslem's efforts on refining the specifications; Jerry Marlatt's work on the preprocessor; and Bill Josephs' development of test cases and assistance in debugging the new commands contributed significantly to a successful development effort. Finally, our appreciation to Roger Fajman of NIH for an advance copy of the external design specifications for NIH Extended WYLBUR.

COMPUTATION CENTER DOCUMENTATION SERIES:

R-1555/1	The Rand Computation Center: Overview and Procedures
R-1555/2	The Rand Computation Center: Language Processors
R-1555/3	The Rand Computation Center: Data Management, Reporting, and Analytical Packages
R-1555/4	The Rand Computation Center: Utilities
R-1555/5	The Rand Computation Center: Subroutines and Rand-Tailored Utilities
R-1555/6	The Rand Computation Center: WYLBUR Learner's Guide
R-1555/7	The Rand Computation Center: WYLBUR Reference Manual
R-1555/8	The Rand Computation Center: Network Access Guide
R-1555/9	The Rand Computation Center: JOSS Users' Reference Manual
R-1555/10	The Rand Computation Center: WYLBUR Command Facilities

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

INTRODUCTION

This manual describes facilities that extend WYLBUR's capabilities in the direction of a programming language that supports structured text entry and text manipulation applications. Using the extensions described here and the facilities described in the WYLBUR Reference Manual (R-1555/7), a user can construct a WYLBUR program to do such things as:

- o Administer a test or questionnaire
- o Prompt for information and construct JCL
- o Format or manipulate a WYLBUR file

Two facilities support these capabilites, they are:

1. Additional WYLBUR commands to permit the following:

   - o The assignment of string and numeric values to variables
   - o Expressions that include arithmetic, logical and relational operators.
   - o Branching based on the value of an expression.
   - o Programmer defined prompts requesting terminal input.
   - o Debugging aids for WYLBUR command programs.

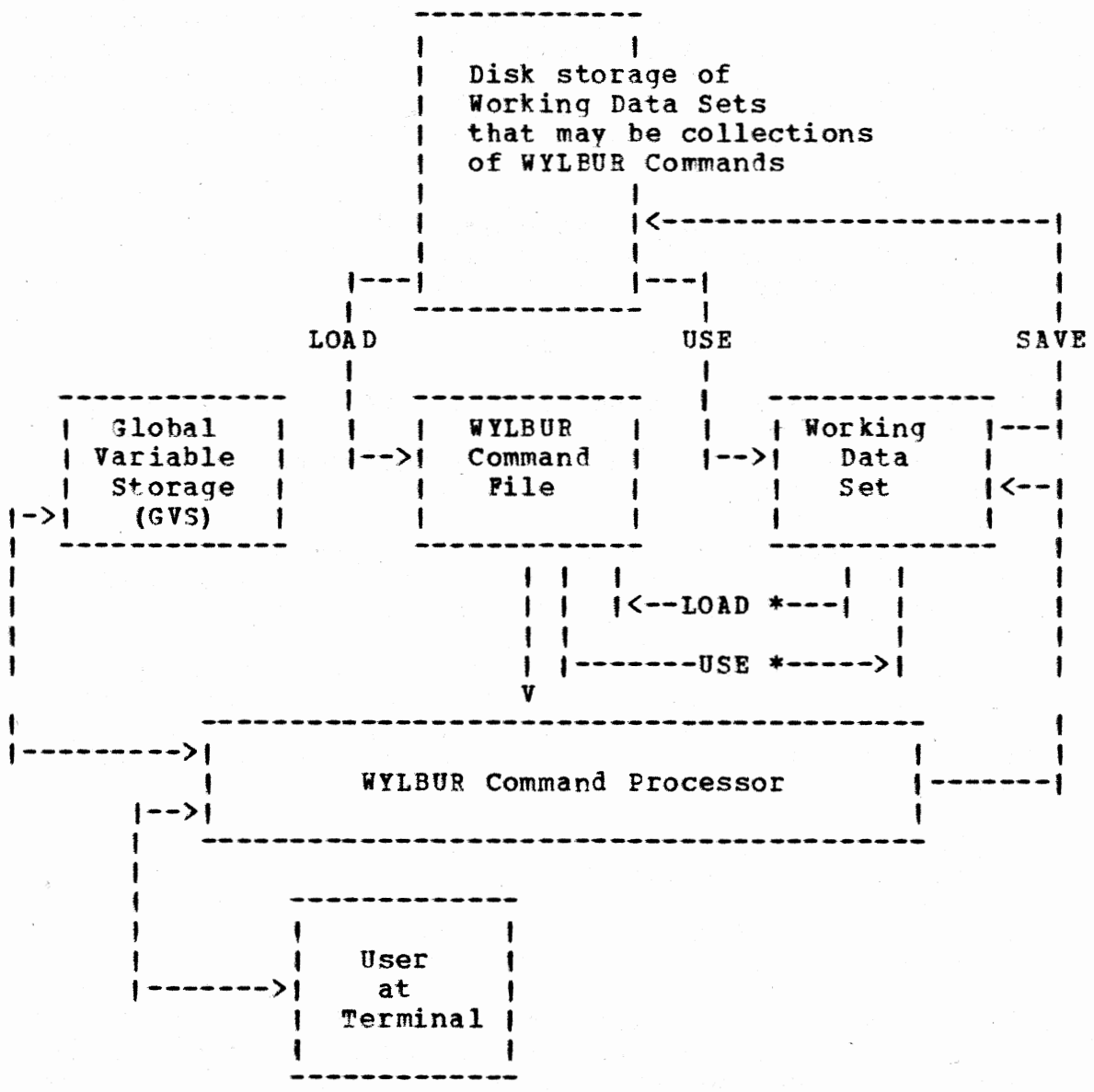
2. A batch preprocessor that extends the language and permits the programmer to write WYLBUR programs in a PL/I format. The WYLBUR program preprocessor supports program logic structure not directly expressible in terms of WYLBUR commands, such as:

   - o Alphanumeric statement labels
   - o Psuedo "automatic" variables
   - o Arrays
   - o Block structured control statements
   - o Subroutine calls

This manual is divided into three sections. Section 1 describes the language elements that support the capability to execute WYLBUR command programs. Section 2 details additional commands and related facilities. Section 3 presents the language understood by the WYLBUR batch preprocessor. Functions are described in Section 1.0, and programming examples are given in Sections 2.2 and 3.4.

There are two prerequisites for understanding this material: 1) familiarity with the material covered in the WYLBUR Reference Manual, R-1555/7, and 2) a reasonable understanding of the art of programming logic.

STRUCTURE OF WYLBUR COMMAND FACILITIES (WCF)



WYLBUR executes commands from two sources: 1) directly from a user at a terminal, or 2) indirectly through the execution of commands stored in a WYLBUR command file. The command file is loaded by issuing a LOAD command in the same manner that a working data set is loaded by issuing a USE command. The LOAD command does not affect the contents of the working data set nor that of Global Variable Storage (GVS). The command file is composed of any commands that can be entered by a user from a terminal. This includes all of the commands that operate on the text of the working data set, commands that USE and SAVE the working data set, submit jobs, etc., plus the command to LOAD the command file itself.

The command file will accommodate approximately 600 WYLBUR commands of average complexity. Programs supporting applications requiring a larger file should be structured to permit overlays of the command file. One of the options of the LOAD command permits the programmer to pass one argument between successive LOADs.

The programmer is permitted to define variables that assist the WYLBUR command program to 'remember' conditions and to follow particular execution sequences. These variables are stored in an area called Global Variable Storage (GVS). GVS can contain up to 5850 characters, which compose a mixture of variable names and their contents. GVS is empty at the beginning of a terminal session. Variables and their contents become defined at their first reference in a WYLBUR command. The contents of GVS is available to successive LOADs within a terminal session. This is similar in concept to the COMMON block in FORTRAN or the EXTERNAL variable declaration in PL/I.

1.0 ELEMENTS OF THE LANGUAGE

CONSTANTS

A constant is a fixed, unvarying quantity. There are three classes of constants:

1) those that specify numbers (numeric constants)
2) those that specify hexadecimal numbers (hexadecimal constants)
3) those that specify character strings (string constants)

Numeric constants are real decimal numbers, hexadecimal constants are real hexadecimal (base 16) numbers, and string constants are a string of alphanumeric and/or special characters.

REAL CONSTANTS

Definition
Real Decimal Constant -- has one of three forms: 1) a basic real decimal constant (a string of decimal digits) 2) a basic real decimal constant followed by a decimal exponent 3) an integer constant Magnitude: 10**-64 < |value| < 10**65 or value=0 Precision: 13 decimal digits

A real constant may be positive, zero, or negative (if unsigned and nonzero, it is assumed to be positive) and must be within the allowable range. It may not contain embedded commas. A zero may be written with a preceding sign, which does not affect the value zero. The decimal exponent permits the expression of a real constant as the product of a basic real constant or integer constant times 10 raised to a desired power.

Examples

VALID REAL CONSTANTS

```
0
-91
173
+0.
-999.9999
7.0E+0
19761.25E+1
7.E3
7.0E3
1.2E+0064
7.0E+03
7E-03
-7E-04
21.98753829457
145789193210984   (Exceeds 13 decimal digits.  The result will
                  be truncated to 13 decimal digits)
```

INVALID REAL CONSTANTS

```
3,471.1           (Embedded comma)
1.E               (Missing a 1- or 2-digit integer
                     constant following the E.)
1.2E+113          (Value exceeds allowable range)
23.5E+97          (Value exceeds allowable range)
21.3E-90          (Value too small)
```

HEXADECIMAL CONSTANTS

Definition
Hexadecimal Constant: the character # followed by a hexadecimal number formed from the set 0 through 9 and A through F.
Magnitude: 0 through FFFFFFFF or (2**32)-1
Precision: 8 hexadecimal digits

Hexadecimal constants are stored as real decimal constants. The HEX function can be used to display real decimal numbers in hexadecimal.

Examples

```
#098FA4
#FF
#EF
```

STRING CONSTANTS

Definition
String Constant: a string of characters of alphabetic, numeric, and/or special characters, enclosed in quotation marks.

The number of characters in the string, including blanks, may not be greater than 255. Since quotation marks delimit the string, a single quotation mark within the string must be represented by two quotation marks. The length of a null string is zero.

Examples

```
'DATA'
'X-COORDINATE        Y-COORDINATE        Z-COORDINATE'
'3.14'
'DON''T'
''                   (null string)
```

SYMBOLIC NAMES

Definition
Symbolic Name: from 1 through 9 letters (A,B,...Z), numbers (0,1,...9), or selected special characters (#,$,@,%)
The first character must be a period and the second must be a letter.

Symbolic names are used in a program to identify variables.

NOTE: The WCFX preprocessor generates variable names containing the special character '@' and, therefore, programmers should refrain from using '@' in their own variable names.

VARIABLES

A WYLBUR variable is a data item, identified by a symbolic name, that occupies space in GVS (analogous to COMMON in FORTRAN). The value specified by the name is always the current value stored in GVS. For example, in the statement

```
LET .A=5.0+.B
```

both .A and .B are variables. The value of .A depends on .B and is calculated when this statement is executed.

An undefined variable (one that has not appeared on the left side of a LET command) has a default value of a null string. In the above example, if the value of .B had not been determined by some previously executed statement, the value of .A after execution of the example statement would be 5. (See Implicit Type Conversion.)

VARIABLE_NAMES

WYLBUR variable names must follow the rules governing symbolic names. Lower case characters (a-z) used to name or refer to a variable are equivalent to the corresponding upper case character. For example, .AbC refers to the variable .ABC.

Examples

VALID VARIABLE NAMES

```
.B292S
.RATE
.VAR$
.Person          (Refers to variable name .PERSON)
```

INVALID VARIABLE NAMES

```
.B292704BCD      (Contains more than 9 characters)
.4ARRAY          (Second character is not alphabetic)
.SI.X            (Contains an illegal special character)
USER             (Does not start with a period)
```

VARIABLE_TYPES

Variables are typed as either character strings or real decimal numbers according to last assignment. See section on IMPLICIT TYPE CONVERSION.

FUNCTIONS

General Form
function-name [(arg1,arg2,....,argn)] Commas are required to separate multiple arguments.

The underlined portion of a function name indicates the permissible abbreviation of that name. The following notation is used to describe the types of arguments permitted with each of the functions supported by the WYLBUR Command Facilities.

c Any single character

i Must be whole integer number <= (2**32-1)

n Any valid real decimal number

s Any valid string

v Must be a variable name

| OR

FUNCTION	ARGUMENT(s)	DESCRIPTION
ABS	n	Returns absolute value of n.
ACCOUNT	none	Returns string equal to account number to which session is being charged.
ARG	none	Returns contents of system argument value, set by using LOAD command with the WITH option.
ATTNLINENO	none	Returns line number at which user pressed the BREAK/ATTENTION key. (This function has not been implemented in the current version of WYLBUR)
BYTE	c	Returns integer equal to internal EBCDIC character representation in decimal.

FUNCTION	ARGUMENT(s)	DESCRIPTION
CASE	none	Returns character string "UPLOW" if uplow option is in effect; otherwise, returns "UPPER".
CEILING	n	Returns smallest integer greater than n.
CHAR	i	For i between 0 and 255, returns corresponding EBCDIC character.
COLUMNS	s,i	Returns substring of string s starting at character position i through the end of s. Returns null string if i is greater than the length of s.
	s,i1,i2	Returns substring of s starting at character position i1 and ending at character position i2. Returns blank string of length i2 if i1 is greater than the length of s.
CONTIME	none	Returns number equal to the terminal connect time in seconds; accurate to 2 decimal places.
CURRENT	none	Returns number equal to the current line number in the working data set.
CPUTIME	none	Returns number equal to WYLBUR editing time in seconds; accurate to 2 decimal places.
DATE	none	Returns character string equal to current date in the form mm/dd/yy.
DEBLANKL	s	Removes blanks from left side of s.
DEBLANKR	s	Removes blanks from right side of s.
DELTA	none	Returns value of global DELTA.
DUPLICATE	s,i	String s is duplicated i times. If the resulting string is longer than 255 characters, it will be truncated.

FUNCTION	ARGUMENT(s)	DESCRIPTION
END	none	Returns number equal to the line number END in the working data set. END is the line number following the last (highest numbered) line in the working data set.
ERRLINENO	none	Returns the line at which last error occurred. (This function has not been implemented in the current version of WYLBUR.)
EVAL	s	Causes s to be evaluated as an expression. Any error in the expression will result in a diagnostic message.
FIRST	none	Returns number equal to the FIRST line number in working data set.
FLOOR	n	Returns the largest integer less than or equal to n.
HEX	i	Returns the hexadecimal character string representation of the integer i.
	s	Returns the hexadecimal character string representation of the internal format of string s.
	i1,i2	Returns the hexadecimal character string representation of i1 padded on the left with blanks to equal the length of i2.
HEXZ	i1,i2	Returns the hexadecimal character string representation of the internal format of i1 padded on the left with zeros to equal the length of i2.
LENGTH	none	Returns integer equal to value entered with previous SET LENGTH command.
	s	Returns integer equal to length of string s.
	n	Returns integer equal to length of line number n in working data set. Returns a -1 if the line does not exist.

FUNCTION	ARGUMENT(s)	DESCRIPTION
INDEX	s1,s2	Returns an integer number representing the starting column of substring s2 in string s1. The search is performed left to right Returns an integer number 0 if s2 is not a substring of s1.
ISNUMBER	v	Returns integer number 1 if v is a number; 0 if v is a string; -1 if v is a string containing a numeric value.
ISSTRING	v	Returns integer number 1 if v is a string; 0 if v is a number.
JDATE	none	Returns character string equal to the Julian date in the form yy.ddd.
LAST	none	Returns number equal to the LAST line number in working data set.
LINE	n	Returns string representing the contents of line n in the working data set. Returns a null line if n does not exist.
LOWER	s	Converts s to all lower case characters.
MAP	s1,s2	s2 is a string containing ordered pairs of characters. String s1 is searched for any occurrence of the first character of any ordered pair in s2. If found, the character in s1 is replaced with the second character of that ordered pair.
NUMBER	s	Returns string s converted to a number. Any error in conversion will result in an error message.
PADL	s,i	String s is padded on the left with blanks to equal length i.
	s1,i,s2	String s1 is padded on the left with characters from string s2 to equal length i. If s2 is shorter than the length i-s1, then s2 will be repeated left to right until i is satisfied.

FUNCTION	ARGUMENT(s)	DESCRIPTION
PADR	s,i	String s is padded on the right with blanks to equal length i.
	s1,i,s2	String s1 is padded on the right with characters from string s2 to total length i. If s2 is shorter than the length i-s1, then s2 will be repeated left to right until i is satisfied.
PAGECOUNT	none	Returns integer equal to number of pages in the working data set.
PREFIX	none	Returns string equal to SET PREFIX. If the prefix has not been set, a null string is returned.
QUOTE	s	String s is quoted. Any internal quotation marks are automatically doubled.
	s1,s2	String s1 is properly quoted using string s2 instead of a quotation mark. String s2 must be of length 1.
ROUND	n	Converts n to an integer by rounding.
	n,i	Returns n rounded to a decimal number with i decimal digits. i may be positive or negative. For example, type round (123.75,0) = 124 t rou (123.75,1) = 123.7 t rou (123.75,-1)= 120 t rou (123.75,-2)= 100
SIZE	none	Returns an integer equal to the number of lines in the working data set.
SIGN	n	Returns integer 1 if n is positive; 0 if n is zero; and -1 if n is negative.

FUNCTION	ARGUMENT(s)	DESCRIPTION
STRING	n	Converts n to a string.
	n,1	Converts n to a string of length i, padding with blanks on the left.
	n,i1,i2	Converts n to a string of length i1, with n truncated or expanded to i2 decimal places to the right of the decimal point.
STRINGZ	n,i	Converts n to a string of length i, with zeros padded on the left.
	n,i1,i2	Converts n to a string of length i1, with n truncated or expanded to i2 decimal places to the right of the decimal point. Zeros are padded on the left as required.
SUBSTITUTE	s	Performs substitution of all variables preceded by "%" and "%%".
	s1,s2,s3	All occurrences of string s2 in string s1 are replaced by string s3.
SUBSTRING	s,i	Returns substring of s starting at character i of s through end of s. Null string is returned if i is greater than the length of s.
	s,i1,i2	Returns substring of s starting at i1 of s for i2 characters. If i1 is greater than the length of s, a string of i2 blanks is returned. The returned string is padded with blanks on the right if necessary.
TERSE	none	Returns string "TERSE" if SET TERSE is in effect; otherwise, the string "VERBOSE" is returned.
TERMINAL	none	Returns character string equal to line address of the terminal.

FUNCTION	ARGUMENT(s)	DESCRIPTION
TERMTYPE	none	Returns integer representing type of terminal in use, e.g., 0 Ann Arbor 4 Anderson Jacobson 8 LSI 12 Memorex 1240 16 Teletype Model 33 20 Teletype Model 37 24 Experimental 28 Texas Instruments 32 (Reserved for future use) 36 GSI-300
TIME	none	Returns character string equal to the current time of day in the form hh:mm:ss AM|PM.
TIME24	none	Returns character string equal to the current time of day, using 24 hour time in the form hh:mm:ss.
TIME100	none	Returns number equal to the current time of day in seconds. Accurate to 2 decimal places.
TRUNCATE	n	Number n is converted by truncation to an integer.
TRUNCATE	n,i	Number n is truncated to i decimal places; i may be positive or negative. For example, Type truncate (123.75,0) = 123 t tru (123.75,1) = 123.7 t tru (123.75,-1)= 120 t tru (123.75,-2)= 100
UPPER	s	String s is converted to all upper case.
USER	none	Returns string equal to user-id to which session is being charged.

FUNCTION	ARGUMENT(s)	DESCRIPTION
VERBOSE	none	Returns string "TERSE" if SET TERSE is in effect; otherwise, the string "VERBOSE" is returned.
VOLUME	none	Returns string equal to volid entered in SET VOLUME command. If no volume has been set, a null string is returned.

CHARACTER CODE GENERATING FUNCTIONS

The following functions have no arguments and a hexadecimal character code is returned for the named character.

FUNCTION NAME	CODE RETURNED (HEX)
BAR	4F
CENT	4A
DEGREE	AF
GREATER	6E
HYPHEN	BF
LEFTCURLY	8B
RIGHTCURLY	9B
LEFTSQUARE	AD
RIGHTSQUARE	BD
LESS	4C
NOTSIGN	5F
PLUSMINUS	9E
UNDERSCORE	6D
UPARROW	8F

PSEUDO-FUNCTIONS

The following functions are valid on the left side of an equal sign in a LET statement.

FUNCTION	ARGUMENT(s)	DESCRIPTION
ARG	none	Value is assigned to the system argument variable. (See LOAD command, WITH option.)
COLUMNS	v,i	The character string defined by v is replaced starting at character position i with the string defined on the right hand side of the equal sign in the LET statement. Replacement terminates when enough characters have been replaced to satisfy the original length of string v. If the length of source string v is less than i+1, it will be treated as if it were padded on the right with the required number of blanks.
COLUMNS	v,i1,i2	The character string defined by v is replaced starting at character position i1 and ending at character position i2 with the string defined on the right-hand side of the equal sign in the LET statement. Blanks are substituted if the length of the source string is less than (i2-i1+1). Resulting string can be longer than the original string.

```
let .v = 'abcdefg'
let col(.v,2,4) = '123'
type .v
A123EFG
let col(.v,2,10) = '123'
type .v
A123␣␣␣␣␣␣␣
let .v = 'abcdefg'
let col(.v,2,3) = '123'
type .v
A12DEFG
```

FUNCTION	ARGUMENT(s)	DESCRIPTION
LINE	n	The line specified by line number n is replaced by the string defined on the right-hand side of of the equal sign in the LET statement. If the line specified by n does not exist, it is created.
SUBSTRING	v,i	(Same as COLUMNS v,i)
	v,i1,i2	The character string defined by v is replaced starting at character position i1 for i2 characters with the string defined on the right-hand side of the equal sign in the LET statement. Blanks are substituted if the length of the source string is less than i2.

1.1 EXPRESSIONS

WYLBUR expressions are composed of a combination of language elements and operators. The language elements can be function calls, variables and constants. Operators can be arithmetic, logical, and relational. Nesting of expressions is limited by statement complexity and available stack space. A diagnostic is issued if stack resources are exceeded.

ORDER OF EXPRESSION EVALUATION

Expression evaluation is performed according to the hierarchy of operations shown in the following list. Within a hierarchial level, computation is performed from left to right.

Hierarchy	Operation
1st	Evaluation of functions
2nd	**
3rd	*,/,//,MOD
4th	+,-
5th	||
6th	GT,NGT,GE,NGE,EQ,NEQ,LT,NLT,LE,NLE
7th	NOT
8th	AND
9th	OR

Symbol	Meaning
Functions	(See Section 1.0)
**	Exponentiation
*	Multiplication
/	Division
//	Truncation after division
MOD	Remainder after division
+	Addition
-	Subtraction
||	String concatenation
GT	Greater than
NGT	Not greater than
GE	Greater than or equal to
NGE	Not greater than or equal to
EQ	Equal to
NEQ	Not equal to
LT	Less than
NLT	Not less than
LE	Less than or equal to
NLE	Not less than or equal to
NOT	(NOT .A) If .A is true then NOT .A has the value false; if .A is false then NOT .A has the value true.
AND	(.A AND .B) If both .A and .B are true, then .A AND .B has the value true; if either .A or .B (or both) is false, then .A AND .B has the value false.
OR	(.A OR .B) If either .A or .B or both are true, then .A OR .B has the value true; if both .A and .B are false, then then .A OR .B has the value false.

The only valid sequences of two logical operators are AND NOT and OR NOT; the sequence NOT NOT is invalid.

If a variable contains a number whose value is non-zero, then that variable is considered true; if the value of the number is zero, then the variable is considered false. If a variable contains a string, it is considered false when the string is null, and true when the string is non-null.

IMPLICIT TYPE CONVERSION

Arithmetic. When evaluating part of an expression involving an arithmetic operator and a string, the string is converted to a number; a diagnostic will result if conversion to a valid number is not possible. The following table illustrates the possible conversions.

Character		Number
'0'	--->	0
''	--->	0
'0'	<---	0

Examples

stringa + numberb	stringa converted to number, if possible.
stringa + stringb	Both strings converted to numbers, if possible.

Concatenation. Numbers are converted to 20 character strings, right justified with leading blanks if required. The sign, if negative, is placed at the character position just prior to the leftmost significant digit.

Examples

stringa||stringb	No conversion
stringa||numberb	numberb converted to 20-character string.
numbera||numberb	Both numbera and numberb converted to 20-character strings.

Relational. The shorter string is padded with trailing blanks to the length of the longer string. When evaluating the relationship between a string and a number, the string is converted to a number; a diagnostic will result if conversion to a valid number is not possible.

Examples

numbera<REL>numberb	No conversion
stringa<REL>stringb	Shorter string will be padded with trailing blanks so that it equals the length of the longer string.
stringa<REL>numberb	stringa is converted to number, if possible.

where <REL> is any of the relational operators, (i.e., GT, NGT,....,NLE).

INDIRECT SUBSTITUTION

Use of "%". When a "%" precedes a language element (i.e., a constant, a variable name, or a parenthesized expression) that element will be replaced with its contents.

Example 1

```
CHANGE 'text' to 'newtext' in %.C
```

will change the string 'text' to 'newtext' in the line(s) specified by the contents of the variable .C.

Example 2

```
COMMAND ? let .x1 = '.x2'
COMMAND ? let .x2 = 'value'
COMMAND ? type .x1
.X2
COMMAND ? type .x2
VALUE
COMMAND ? type %.x1
VALUE
```

Example 3

```
COMMAND ? let .a2 = 'string'
COMMAND ? let .b = '2'
COMMAND ? let .c = %('.a'||.b)
COMMAND ? type .c
STRING
```

will assign to variable .C the value of variable .A2 (e.g., the value 'STRING').

Use of "%%". When a "%%" precedes a language element (e.g., a constant, a variable name, or a parenthesized expression) that element will be replaced by its contents, and the resulting value will be enclosed in quotation marks.

Example 1

```
COMMAND ? set uplow
COMMAND ? let .a = 'text'
COMMAND ? let .b = 'newtext'
COMMAND ? let .c = '1/10'
COMMAND ? change %%.a to %%.b in %.c
```

will result in the following command:

```
    CHANGE 'text' TO 'newtext' IN 1/10
```

Example 2

```
COMMAND ? let .a = 'test'
COMMAND ? let .b = 'exam'
COMMAND ? request str .a prompt %%.b
EXAM
```

will request a user-supplied value to be assigned to variable .A by prompting with the contents of the variable .B.

Example 3

```
COMMAND ? let .cur = 13403
COMMAND ? l %%(str(.cur,8))
```

will result in the following command:

```
L '00013403'
```

RESTRICTIONS

Indirect substitution as described above operates only at the first level of reference. In Example 1 above, if the contents of .C had been %.D then the substitution would have resulted in the following:

```
CHANGE 'text' TO 'newtext' IN %.D
```

Since %.D would require a second level substitution, the command would be illegal.

Indirect substitution can not be used within a parenthesized expression. This prevents the use of indirect substitution as an argument for a function call or for nesting indirect substitutions within an expression.

Thirdly, indirect substitution may not be used as a pseudo-variable in an assignment statement.

2.0 COMMAND STATEMENTS

This section defines the conventions used in presenting WYLBUR commands.

- Words printed in upper case are keywords and should be entered as shown.
- Permissible abbreviations of keywords are indicated by underlining.
- Lower case words are replaced by user-supplied values.
- Words and phrases bounded by '{' and '}' are mandatory within a command.
- Words and phrases bounded by '[' and ']' are optional within a command depending upon the function desired.
- '|' within the definition of a command means 'OR'. For example, the text "constant|varname|expression" should be read, "constant or variable name or expression".
- 'Φ' within the definition of a command means 'AND OR'.

PROGRAM LOADING STATEMENTS

LOAD Command

General Form
LOAD {dsname|*} [ON volid] [LIST] [GO [lineno]] [WITH constant|varname|(expression)] [CLEAR]

The LOAD command copies the contents of the specified disk data set into the WYLBUR command file. A LOAD * command transfers the contents of the working data set into the command file. The working data set is cleared by the action of the LOAD * command.

dsname — is the name of the data set containing the WYLBUR command file to be loaded for execution.

* — is the working data set that is cleared by the LOAD * command.

ON volid — is a keyword followed by the name of the disk volume where the data set is stored. Not required if the dsname is cataloged. May not be specified when the * option is used.

LIST — is a keyword option causing each command executed from the command file to be listed at the terminal prior to execution.

GO [lineno] — is a keyword option allowing specification of the line number of the WYLBUR command to be executed after loading is complete. If this option is omitted, control will be returned to the terminal after loading. If the optional line number is omitted, execution will start at the first (lowest numbered) statement in the command file.

WITH arg — is a keyword followed by an argument that may be any one of the indicated language elements. The value of the element is stored as a string in GVS and is accessible through use of the ARG function. (See FUNCTIONS in Section 1.0.)

CLEAR — is a keyword option that is required if the command file has been loaded previously during the terminal session. If this option is omitted, and the command file is not empty, a prompt will

be issued at the terminal requesting permission to clear the command file so that loading can take place.

Example 1

```
COMMAND ? load command.file on user03 clear
```

Example 2

```
COMMAND ? load commands clear go
THIS COMMAND FILE HAS STARTED
          .
          .
          .
```

Example 3

```
COMMAND ? load @member with .parm clear
```

Example 4

```
COMMAND ? load set.commands clear go list
   1.     -> SET TERSE
   2.     -> SET UPLOW
   3.     -> SET VOL USER01
   4.     -> SET NOBREAK
```

Example 5

```
COMMAND ? collect
   1.    ? set tabs 5 10 20 30 40 50
   2.    ? set terse
   3.    ? set vol user02
   4.    ? set uplow
   5.    ? ***                (= BREAK/ATTENTION key pressed)
COMMAND ? load * clear go
LOG. TABS      1    1          1          1          1          1
PHY. TABS      1    1          1          1          1          1
```

Example 6

```
COMMAND ? use command.file clear
COMMAND ? delete 100/last
COMMAND ? load * clear go
THIS COMMAND FILE HAS BEGUN
          .
          .
          .
```

USE * Command

General Form
USE * [CLEAR]

The USE * command transfers the contents of the command file to the working data set. The command file is cleared by the action of the USE * command.

* represents the current contents of the command file which is cleared by this command.

CLEAR is a keyword option that is required if the WYLBUR working data set contains any lines. If this option is omitted, and the working data set is not empty, a prompt will be issued at the terminal requesting permission to clear the working data set so that the contents of the command file can be transferred to it.

Example 1

```
COMMAND ? load command.file clear go
FUNC - UNKNOWN FUNCTION
COMMAND ? use * clear
COMMAND ? mod 'func' in all
          .
          .
          .
```

Example 2

```
COMMAND ? collect
   1.   ? set vol user10
   2.   ? set nobreak
   3.   ? set uplow
   4.   ? ***                    (= BREAK/ATTENTION key pressed)
COMMAND ? load * clear go
USER10 - ILLEGAL VOLUME
COMMAND ? use *
COMMAND ? change 'user10' to 'user01' nolist
COMMAND ? load * go
```

ASSIGNMENT STATEMENTS

LET Command

General Form
LET {pseudo-function-list@variable-list} = expression

The LET command assigns the value of the expression to the named variable or pseudo-function. Multiple assignments are allowed but variable or pseudo-function lists must be enclosed in parentheses.

Examples

```
COMMAND ? let .a = 1


COMMAND ? let (.a,.b) = 4.57


COMMAND ? let .c = 'string'


COMMAND ? let .str='1234567890'
COMMAND ? let substr(.str,3,5) = '*****'
COMMAND ? type .str
12*****890


COMMAND ? clear text
COMMAND ? let line(1) = 'this is line one'
COMMAND ? list
   1.     THIS IS LINE ONE


COMMAND ? let .value = (.num1**3/2.54)+((.num2+4)//.num3)


COMMAND ? let .str = 'this'||' is a '||'string'


COMMAND ? let .a = '+'
COMMAND ? let .a= .a||strz(cputime)
COMMAND ? type .a
+.37
COMMAND ? let substr(.a,2,4)=account
COMMAND ? type .a
+2093
```

SHOW GLOBALS Command

General Form
SHOW GLOBALS [varname-list]

The SHOW GLOBALS command converts the values of the named variables to printable form and displays both names and values at the terminal. If a variable name list is not supplied, all defined variables will be displayed.

Example 1

```
COMMAND ? show globals
NO GLOBAL VARIABLES DEFINED
COMMAND ? let (.a,.b) = 'value'
COMMAND ? show globals
.A          = VALUE
.B          = VALUE
```

Example 2

```
COMMAND ? let .c = 24
COMMAND ? show globals .c
.C          =                    24
COMMAND ?
```

CLEAR GLOBALS Command

General Form
CLEAR GLOBALS [varname-list]

The CLEAR GLOBALS command removes the definitions and values of all named variables from GVS. If the variable name-list is omitted, all user-defined variables will be removed from GVS.

Example 1

```
COMMAND ? show globals
.A          = VALUE
.B          = VALUE
COMMAND ? clear globals
COMMAND ? show globals
NO GLOBAL VARIABLES DEFINED
COMMAND ? show globals .a
.A          =
```

Example 2

```
COMMAND ? show globals
.A          = STRING1
.B          = STRING2
COMMAND ? clear globals .a
COMMAND ? show globals
.B          = STRING2
```

BRANCHING STATEMENT

GO Command

General Form
GO [TO] [lineno] [LIST|NOLIST]

The GO command causes execution to proceed from a given line number. If the line number is omitted, execution starts with the first command in the command file or with the next command that would have been executed if execution had previously been interrupted.

LIST — is a keyword option causing each command about to be executed from the command file to be listed at the terminal prior to execution. This option overrides a "NOLIST" entered with a previous GO command.

NOLIST — is a keyword option that overrides a LIST entered with the LOAD command or previous GO command. NOLIST commands are not listed.

At LOAD time the default mode is NOLIST. Thereafter, the option last specified is retained if no option is specified on the current command.

Example 1

```
COMMAND ? load command.file clear
COMMAND ? go
THIS COMMAND FILE HAS BEGUN
          .
          .
          .
```

Example 2

```
COMMAND ? collect
   1.   ? request str .a
   2.   ? if .a eq ''
   3.   ? else goto 5
   4.   ? let .a = ' '
   5.   ? type .a
          .
          .
          .
```

Example 3

```
COMMAND ? load set.commands clear
COMMAND ? go list
   1.     -> SET UPLOW
   2.     -> SET NOBREAK
   3.     -> SET VOL USER05
   4.     -> SET TERSE
```

CONDITIONAL STATEMENTS

IF Command

General Form
IF expression

After the IF command is executed, the value of the internal "IF switch" is set to false if the "expression" was equal to a numeric zero or a null string; otherwise, the value of the switch is set to true.

THEN Command

General Form
THEN command

The THEN command causes the value of the "IF switch" to be tested; if it is true, the command is executed. Otherwise, this command is skipped, and the next command is executed.

ELSE Command

General Form
ELSE command

The ELSE command causes the value of the "IF switch" to be tested, and if false the ELSE command is executed. Otherwise, this command is skipped, and the next command is executed.

Example_1

```
 1.      type 'example'
 2.      request str .a
 3.      if .a eq '1' or .a eq '0'
 4.      .
         .
         .
```

Example_2

```
         .
         .
         .
53.      if substr(.var1||.var2,10,5) eq '*****' or .var1 eq ''
54.      then goto 112
55.      if .value1 ngt cputime
56.      .
         .
         .
```

Example_3

```
         .
         .
         .
10.      if .b neq 0
11.      then let .b = 1
12.      then goto 50
13.      .
         .
         .
```

Example_4

```
         .
         .
         .
77.      if .var1 gt 10
78.      then if .var2 ngt 5
79.      then let .var3 = 15
         .
         .
         .
```

Example 5

```
          .
          .
          .
15.    ? if .c gt 0
16.    ? else let .c = 0
17.    ? else goto 25
18.    ? .
          .
          .
```

Example 6

```
          .
          .
          .
23.       if length(.string) nlt .maxlen
24.       else let .string = .string||.char
25.       else goto 53
          .
          .
          .
```

INPUT STATEMENT

REQUEST Command

General Form
REQUEST {STRING|NUMBER|VALUE} varname [PROMPT {constant|variable|(expression)}] [ATTENTION lineno] [DEBLANK {LEFT|RIGHT}]

The REQUEST command prompts the user with a varname to which he must assign a value. The user must specify one of the following elements in the command: STRING, NUMBER, or VALUE.

STRING — treats the response entered as a character string which takes the value of the variable.

NUMBER — treats the response entered as a numeric constant.

VALUE — treats the response entered as an expression, which is evaluated and stored as a value.

varname — is the name of the variable in GVS which will be assigned the value of the response that has been entered.

PROMPT — allows specification of a string to prompt for the response to be entered. If the PROMPT parameter is omitted, the name of the variable (varname) will be used.

ATTENTION — specifies the line number of a command that will receive control if the BREAK/ATTENTION key is struck without a preceding keystroke, i.e., while responding to a REQUEST prompt.

DEBLANK — removes blanks from the right or left margins of the response.

Example 1

```
COMMAND ? request string .a
.A = this is a string
COMMAND ? type .a
THIS IS A STRING
```

Example 2

```
COMMAND ? collect
  1.    ? request string .a prompt 'value? ' deblank left
  2.    ? request string .b prompt '2nd value? ' deblank left
  3.    ? type .a||.b
  4.    ? ***                          (= BREAK/ATTENTION key pressed)
COMMAND ? load * go clear
VALUE? 12345
2ND VALUE?    abcde
12345ABCDE
```

OUTPUT STATEMENTS

TYPE Command

General Form
TYPE expression

The TYPE command evaluates the "expression", converts it to printable form, and displays it at the terminal.

Example 1

```
COMMAND ? type 'this is an example'
THIS IS AN EXAMPLE
```

Example 2

```
COMMAND ? let .a = date
COMMAND ? type .a
03/10/75
COMMAND ? type 'the current date is '||.a
THE CURRENT DATE IS 03/10/75
```

Example 3

```
COMMAND ? type case
UPPER
COMMAND ? type 1/3
              .3333333333333
```

Example 4

```
COMMAND ? let .a = 2
COMMAND ? let .b = 1.41421356
COMMAND ? type 'the square root of '||strz(.a)||' is '||strz(.b)
THE SQUARE ROOT OF 2 IS 1.41421356
COMMAND ? type 'the square of '||strz(.b)||' is '||strz(.b*.b)
THE SQUARE OF 1.41421356 IS 1.99999999328
```

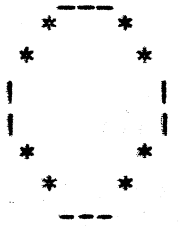
TTYPE Command

General Form
TTYPE expression

The TTYPE command evaluates the "expression", and transmits it to the terminal without any intervening code translation. The evaluated expression is assumed to result in a string of characters in line code. (See TTYPE TRANSPARENCY TABLE.)

Example

```
COMMAND ? collect
   1.   ? ttype char(#31)
   2.   ? ttype char(#f0)||char(#00)||char(#82)
   3.   ? ttype dup(char(#4),20)||dup(char(#fb),3)||char(#b1)||
               char(#51)
   4.   ? ttype dup(char(#4),19)||char(#55)||dup(char(#4),3)||
               char(#55)||char(#b1)||char(#51)
   5.   ? ttype dup(char(#4),18)||char(#55)||dup(char(#4),5)||
               char(#55)||char(#b1)||char(#51)
   6.   ? ttype dup(char(#4),17)||char(#3f)||dup(char(#4),7)||
               char(#3f)||char(#b1)||char(#51)
   7.   ? ttype dup(char(#4),17)||char(#3f)||dup(char(#4),7)||
               char(#3f)||char(#b1)||char(#51)
   8.   ? ttype dup(char(#4),18)||char(#55)||dup(char(#4),5)||
               char(#55)||char(#b1)||char(#51)
   9.   ? ttype dup(char(#4),19)||char(#55)||dup(char(#4),3)||
               char(#55)||char(#b1)||char(#51)
  10.   ? ttype dup(char(#4),20)||dup(char(#b5),3)||char(#b1)||
               char(#51)
  11.   ? ***                    (= BREAK/ATTENTION key pressed)
COMMAND ? load * go
```



DEBUGGING STATEMENTS

COMMAND Command

General Form
COMMAND

The COMMAND command is a debugging aid that can be included in the command file program to break execution and optionally display information. Following execution of a COMMAND command, WYLBUR prompts with "COMMAND ?" signifying that the next command entry is expected from the terminal. Execution from the command file will be resumed automatically following the entry of one command from the terminal.

Example

```
COMMAND ? collect
   1.   ? let .a = 1
   2.   ? let .b = 2
   3.   ? let .c = 3
   4.   ? command
   5.   ? type strz(.a)||strz(.b)||strz(.c)
   6.   ? ***                    (= BREAK/ATTENTION key pressed)
COMMAND ? load * go clear
COMMAND ? show global
.A          =                    1
.B          =                    2
.C          =                    3
123
```

PAUSE Command

General Form
PAUSE [expression]

The PAUSE command is a debugging aid which can be included in the command file program to break execution and optionally display information. Following execution of a PAUSE command, WYLBUR types the message "PAUSE HAS BEEN EXECUTED" followed by the "COMMAND ?" prompt, signifying that the next command entry is expected from the terminal. Execution from the command file can be resumed following a PAUSE by the command GO in the same manner that execution is resumed after the BREAK/ATTENTION key is struck.

expression — indicates that the specified expression be evaluated, converted to printable form, and displayed as the message preceding the "COMMAND ?" prompt.

Example 1

```
COMMAND ? collect
   1.   ? let .a = 1
   2.   ? let .b = 2
   3.   ? let .c = 3
   4.   ? pause
   5.   ? type strz(.a)||strz(.b)||strz(.c)
   6.   ? ***                    (BREAK/ATTENTION key pressed)
COMMAND ? load * go clear
PAUSE HAS BEEN EXECUTED
COMMAND ? show global
.A          =                    1
.B          =                    2
.C          =                    3
COMMAND ? go
123
COMMAND ?
```

Example 2

```
COMMAND ? collect
   1.   ? type 'first'
   2.   ? pause 'enter go to resume'
   3.   ? type 'last'
   4.   ? ***            (=BREAK/ATTENTION key pressed)
COMMAND ? load * go clear
FIRST
ENTER GO TO RESUME
COMMAND ? go
LAST
COMMAND ?
```

STEP Command

General Form
STEP numeric-expression [FROM lineno] [LIST|NOLIST]

The STEP command is a debugging aid that permits execution of a specified portion of the command file program. It is normally entered directly from the terminal in response to a PAUSE command in the command file program.

num-expr — specifies the number of commands to be executed before the user is asked to respond to a PAUSE command.

FROM — causes execution to resume from the specified line number.

LIST — causes commands to be listed before execution.

NOLIST — causes LIST option to be terminated.

At LOAD time the NOLIST option is in effect. Thereafter, the option last specified is retained if no option is specified on the current command.

Example

```
COMMAND ? collect
   1.   ? let .sum=0
   2.   ? let .i=1
   3.   ? go 5
   4.   ? let .i = .i+1
   5.   ? if .i gt 10
   6.   ? then go 10
   7.   ? let .sqr = .i*.i
   8.   ? let .sum = .sum+.sqr
   9.   ? go 4
  10.   ? type .sum
  11.   ? ***                    (= BREAK/ATTENTION key pressed)
COMMAND ? load * clear
COMMAND ? step 25
STEP COMPLETE, PAUSE EXECUTED.
COMMAND ? show global
.I          =                    4
.SQR        =                   16
.SUM        =                   30
COMMAND ? step 50
```

SET TRACE Command

General Form
SET [TRACE|NOTRACE] [varname-list|VARIABLES]

The SET TRACE command is a debugging tool normally entered directly from the terminal. It is used to trace particular variables as they change during execution of a command file program.

TRACE — turns on the specified trace options. When the "varname-list" is present, and not overridden by the VARIABLES option only the variables specified on the list are displayed when changed.

NOTRACE — turns off specified trace options.

VARIABLES — causes all defined variables to be displayed when changed.

Example

```
COMMAND ? collect
   1.   ? let .sum = 0
   2.   ? let .i = 1
   3.   ? go 5
   4.   ? let .i = .i+1
   5.   ? if .i gt 5
   6.   ? then go 10
   7.   ? let .sqr = .i*.i
   8.   ? let .sum = .sum+.sqr
   9.   ? go 4
  10.   ? type .sum
  11.   ? ***                    (=BREAK/ATTENTION key pressed)
COMMAND ? set trace .sum
COMMAND ? load * clear go
   1.   : .SUM =                  0
   8.   : .SUM =                  1
   8.   : .SUM =                  5
   8.   : .SUM =                 14
   8.   : .SUM =                 30
   8.   : .SUM =                 55
```

2.1 WCF ERROR MESSAGES

o CONVERSION ERROR

An attempt to convert an internal numeric value to a character string has failed.

o err-txt - 2ND ARGUMENT:EVEN NUMBER OF CHARACTERS REQUIRED

The second argument to the function MAP must be a character string containing ordered pairs of characters to be used in the transformation.

o err-txt - 3RD ARGUMENT LESS THAN 2ND

The third argument to the function COLUMNS is less than the second, resulting in an attempt to obtain a string of negative length.

o err-txt - ARGUMENT MUST BE A CHARACTER STRING

The first argument of the pseudo-functions SUBSTRING and COLUMNS must be a variable that contains a string value; otherwise, conversion will not be attempted.

o err-txt - ARGUMENT MUST BE A LINE NUMBER

The argument of the LINE pseudo-function must be a numeric value that can be converted to a valid WYLBUR line number.

o err-txt - ARGUMENT MUST BE POSITIVE INTEGER

An argument for the named function (or pseudo-function) "err-txt" must be a numeric integer value greater than zero.

o err-txt - ARGUMENT MUST BE POSITIVE INTEGER LESS THAN OR EQUAL TO 255

Arqument to the function CHAR must be an integer between zero and 255 inclusive.

o err-txt - CHARACTER MUST BE OF LENGTH ONE

Character to be used for quoting the given string can only be of length one.

o err-txt - FIELD TOO SMALL FOR RESULT

The character string resulting from the conversion of an integer value to a hex string is too large to fit in the supplied return area.

WCF ERROR MESSAGES (continued)

o err-txt - FUNCTION NOT IMPLEMENTED

"err-txt" is a valid function name but is not currently supported.

o err-txt - ILLEGAL ARGUMENT SPECIFICATION

The number of arguments supplied for the function "err-txt" is illegal.

o err-txt - ILLEGAL ARGUMENT TYPE

An argument supplied for the function "err-txt" is of an improper type, i.e., a character string was found instead of a numeric string.

o err-txt - ILLEGAL LINE NO.

Argument to the function LINE or LENGTH must be a numeric value that can be converted to a valid WYLBUR line number.

o err-txt - ILLEGAL VARIABLE NAME

"err-txt" is an illegal variable name.

o err-txt - THIRD ARGUMENT ILLEGAL

The third argument of the pseudo-function COLUMNS is less than or equal to the first argument.

o err-txt - UNKNOWN FUNCTION

"err-txt" is not a function.

o err-txt - UNKNOWN PSEUDO-FUNCTION

"err-txt" is not a pseudo-function.

o ERROR IN NUMERIC CONVERSION

An attempt to convert a string of characters to an internal numeric representation has failed.

o NEGATIVE OR NON-INTEGER POWERS NOT ALLOWED

Exponentiation is legal only for positive integer powers.

WCF ERROR MESSAGES (continued)

o NO MORE ROOM FOR VARIABLES. COMMAND TERMINATED

In an attempt to save a variable in GVS, you have exceeded the maximum amount of space allowed.

o NOT CURRENTLY EXECUTING MACRO

An attempt was made to set up a step debugging environment outside an active use of the WYLBUR Command Facility.

o NUMERIC VALUE REQUIRED

An attempt to enter a non-numeric value in response to a REQUEST command with the NUMBER option has been encountered.

o POSITIVE INTEGER REQUIRED

The STEP command requires that the requested number of steps be an integer greater than zero.

o "STRING", "VALUE", AND "NUMBER" CONFLICT

The command REQUEST requires the specification of one and only one parameter from the group: STRING, NUMBER or VALUE

o WYLBUR OUT OF PAGES. COMMAND TERMINATED

In an attempt to save a variable in the variable work space, WYLBUR has exceeded the capacity of its paging files.

----------NOTES----------

2.2 WCF PROGRAMMING EXAMPLE

```
 1.  CLEAR TEXT
 2.  CLEAR GLOBAL
 3.  TYPE 'IEHMOVE COPY PROMPTER'
 4.  REQUEST STRING .BIN PROMPT 'BIN='
 5.  1 //%(USER)RXX JOB <%ACC,5,%(.BIN)>,COPY,CLASS=G
 6.  2 // EXEC PGM=IEHMOVE
 7.  3 //SYSPRINT DD SYSOUT=A
 8.  4 //SYSUT1 DD UNIT=TEMP,VOL=SER=TEMP10,DISP=SHR
 9.  REQUEST STRING .DSN PROMPT 'DSNAME TO COPY = '
10.  IF SUBSTR(.DSN,1,1) EQ '$'
11.  THEN LET .DSN=SUBSTR(.DSN,2)
12.  ELSE LET .PREFIX=SUBSTR(USER,1,1)||'.'||USER||'.A'||ACC
13.  ELSE LET .DSN=.PREFIX||'.'||.DSN
14.  REQUEST STRING .FROMDEV PROMPT 'ENTER FROM DEVICE '
15.  REQUEST STRING .FROMVOL PROMPT 'ENTER FROM VOLUME '
16.  IF SUBSTR (.FROMDEV,1,4) EQ 'TAPE'
17.  ELSE GO TO 23
18.  REQUEST STRING .FROMFIL PROMPT 'ENTER FROM FILE # '
19.  5 //FROM DD UNIT=%.FROMDEV,LABEL=(1,SL),DISP=(OLD,KEEP),
20.  6 //         VOL=<PRIVATE,RETAIN,SER=00%(.FROMVOL)>
21.  LET .FROMVOL='<'||.FROMVOL||','||.FROMFIL||'>'
22.  GO TO 24
23.  5 //FROM DD UNIT=%.FROMDEV,VOL=SER=&.FROMVOL,DISP=OLD
24.  REQUEST STRING .TODEV PROMPT 'ENTER TO DEVICE '
25.  REQUEST STRING .TOVOL PROMPT 'ENTER TO VOLUME '
26.  IF SUBSTR(.TODEV,1,4) EQ 'TAPE'
27.  ELSE GO TO 33
28.  REQUEST STRING .TOFIL PROMPT 'ENTER TO FILE # '
29.  7 //TO DD UNIT=%.TODEV,LABEL=(1,SL),DISP=(OLD,KEEP),
30.  8 //         VOL=<PRIVATE,RETAIN,SER=00%(.TOVOL)>
31.  LET .TOVOL='<'||,TOVOL||'.'||.TOFIL||'>
32.  GO TO 34
33.  7 //TO DD UNIT=%.TODEV,VOL=SER=%.TOVOL,DISP=OLD
34.  9 //SYSIN DD *
35.  10  COPY DSNAME=%.DSN
36.  CHANGE 72 to 'X' IN 10 NOLIST
37.  11 TO=$.TODEV=%.TOVOL,FROM=%.FROMDEV=%.FROMVOL,TODD=TO
38.  ALIGN 11 INDENT 15
39.  CHANGE '<' TO '(' NOLIST
40.  CHANGE '>' TO ')' NOLIST
41.  LIST
```

Explanation of Commands

1. Clear the working file.
2. Clear any variables.
3. Put herald message on terminal.
4. Ask user for his output bin#.
5. Create JOB card as line 1 in the working data set using logon user-id and account#. NOTE: Use <> instead of () until end since % replacement doesn't take effect inside parens.
6. Create EXEC statement as line 2.
7. Create SYSPRINT DD card as line 3.
8. SYSUT1 DD card as line 4.
9. Ask user for data set name to copy.
10. See if prefixed by $==>full dsname.
11. If $, then must strip $ off dsname.
12. Otherwise, build prefix from logon user-id/account#,
13. and build fully qualified dsname.
14. Ask user for source device name,
15. and source volume.
16. See if source is tape.
17. Skip if not tape.
18. Ask user for tape file#.
19. Create a tape FROM DD statement.
20. Continuation of FROM DD statement.
21. Build volume parm for COPY utility statement to be created later.
22. Skip non-tape DD statement.
23. Create disk FROM DD statement.
24. Ask user for destination device name,
25. and destination volume.
26. See if destination is tape.
27. Skip if not tape.
28. Ask user for tape file#.
29. Create tape TO DD statement.
30. Continuation of TO DD statement.
31. Build volume parm for COPY utility statement to be created later.
32. Skip non-tape DD statement.
33. Create disk TO DD statement.
34. Create SYSIN statement as line 9.
35. Create COPY utility input statement.
36. Put continuation character in column 72,
37. and continue creating COPY statement.
38. COPY statement must continue in column 16.
39. Change all < to (in job stream.
40. Change all > to) in job stream.
41. List job for user to check and run.

Program Output

```
RAND COMPUTATION CENTER   LINE 11  (02B)  10/30/74   2:53:00 P.M.
THE SYSTEM WILL BE DOWN FROM 18:00-18:15 HRS., QUICK RELOAD
USER? h2550
ACCOUNT? ####
KEYWORD? ###
COMMAND ? load copy go
IEHMOVE COPY PROMPTER
BIN=152
DSNAME TO COPY = source.read
ENTER FROM DEVICE tape9
ENTER FROM VOLUME 2387
ENTER FROM FILE # 26
ENTER TO DEVICE 3330
ENTER TO VOLUME user50
   1.      //H2250RXX JOB (2091,5,152),COPY,CLASS=3
   2.      // EXEC PGM=IEHMOVE
   3.      //SYSPRINT DD SYSOUT=A
   4.      //SYSUT1 DD UNIT=TEMP,VOL=SER=TEMP10,DISP=SHR
   5.      //FROM DD UNIT=TAPE9,LABEL=(1,SL),DISP=(OLD,KEEP),
   6.      //      VOL=(PRIVATE,RETAIN,SER=002387)
   7.      //TO DD UNIT=3330,VOL=SER=USER50,DISP=OLD
   9.      //SYSIN DD *
  10.       COPY DSNAME=H.H2250.A2091.SOURCE.READ,                    X
  11.                    TO=3330=USER50,FROM=TAPE9=(2387,26),TODD=TO
COMMAND ? run
```

2.3 TTYPE TRANSPARENCY TABLE

Ann Arbor Line Codes

The following table lists the characters available on Ann Arbor terminals and their corresponding hexadecimal codes. These hexadecimal codes are the reversed ASCII codes sent to Ann Arbor terminals.

A	83	H	13	O	F3	V	6B	0	0D
a	87	h	17	o	F7	v	6F	1	8D
B	43	I	93	P	0B	W	EB	2	4D
b	47	i	97	p	0F	w	EF	3	CD
C	C3	J	53	Q	8B	X	1B	4	2D
c	C7	j	57	q	8F	x	1F	5	AD
D	23	K	D3	R	4B	Y	9B	6	6D
d	27	k	D7	r	4F	y	9F	7	ED
E	A3	L	33	S	CB	Z	5B	8	1D
e	A7	l	37	s	CF	z	5F	9	9D
F	63	M	B3	T	2B				
f	67	m	B7	t	2F				
G	E3	N	73	U	AB				
g	E7	n	77	u	AF				

&	65	Ampersand	<	3D	Less than
*	55	Asterisk	LF	51	Line feed
@	03	At sign	-	B5	Minus sign
DC1	89	Attn/Break	SI	F0	Move cursor
	79	Backspace	(	15	Paren, left
|	3F	Bar, Vertical	)	95	Paren, right
{	DF	Brace, left	%	A5	Percent sign
}	BF	Brace, right	.	75	Period
[	DB	Bracket, left	+	D5	Plus sign
]	BB	Bracket, right	#	C5	Pound sign
CR	B1	Carriage return	?	FD	Question mark
FS	38	Clear screen and tabs	"	45	Quotation mark, double
			'	E5	Quotation mark, single
FF	31	Clear screen but not tabs	'	07	Quotation mark, single reversed
:	5D	Colon	:	DD	Semicolon
,	35	Comma	GS	B8	Set tab
$	25	Dollar sign		3B	Slash, reverse (used as not-sign)
ESC	D9	Escape			
=	BD	Equal sign	/	F5	Slash, standard
!	85	Exclamation mark	HT	91	Tab, horizontal
>	7D	Greater than	VT	D0	Tab, vertical
NUL	01	Idle character	_	FB	Underscore

For 9600 baud lines:

1. FF or FS must be followed by 2 VT's.
2. LF must be followed by 1 NUL.
3. Any transmission that writes a character in the rightmost column and causes the screen to roll, must follow that character with 1 NUL.

Ann Arbor Column and Row Codes

The Ann Arbor cursor may be moved directly to a new position on the screen by sending the SI character (F0) followed by a one-character column code and a one-character row code. The column and row codes are listed below.

Column Codes

0	00	20	04	40	02	60	06
1	80	21	84	41	82	61	86
2	40	22	44	41	42	62	46
3	C0	23	C4	43	C2	63	C6
4	20	24	24	44	22	64	26
5	A0	25	A4	45	A2	65	A6
6	60	26	64	46	62	66	66
7	E0	27	E4	47	E2	67	E6
8	10	28	14	48	12	68	16
9	90	29	94	49	92	69	96
10	08	30	0C	50	0A	70	0E
11	88	31	8C	51	8A	71	8E
12	48	32	4C	52	4A	72	4E
13	C8	33	CC	53	CA	73	CE
14	28	34	2C	54	2A	74	2E
15	A8	35	AC	55	AA	75	AE
16	68	36	6C	56	6A	76	6E
17	E8	37	EC	57	EA	77	EE
18	18	38	1C	58	1A	78	1E
19	98	39	9C	59	9A	79	9E

Row Codes

0	02	10	52	20	06	30	56
1	82	11	D2	21	86	31	D6
2	42	12	32	22	46	32	36
3	C2	13	B2	23	C6	33	B6
4	22	14	72	24	26	34	76
5	A2	15	F2	25	A6	35	F6
6	62	16	0A	26	66	36	0E
7	E2	17	8A	27	E6	37	8E
8	12	18	4A	28	16	38	4E
9	92	19	CA	29	96	39	CE

3.0 INTRODUCTION TO WCFX

WCFX is an extension of the WYLBUR Command Facility (WCF) that assists users in developing WCF programs. The WCFX language provides the WCF programmer with many of the features of high-level languages such as freeform input, alphanumeric statement labels, and block structuring statements, while retaining a simplicity that makes learning and using the language easy.

The WCFX processor accepts a sequence of statements written in WCFX and/or WCF (the entire WCF language is a proper subset of WCFX) and translates it into an equivalent WCF program. The resulting program is then processed by WYLBUR as a standard WYLBUR command file. Corrections to a program may be made at either the WCFX "source" level or the WCF "object" level.

RUNNING WCFX PROGRAMS

The WCFX processor runs as a batch job and produces a WCF program as a final result. The JCL for using WCFX is:

```
//          JOB  statement
//          EXEC WCFX [options]
//SYSIN     DD   *
      (WCFX program)
/*
```

where [options] may include any of the following:

,GOFILE=dsname

The specified file will contain the resulting WCF program on completion; if this option is omitted, no WCF program is produced.

,UNIT=unit

Defines the unit type (e.g., USER, 2314, 3330) on which the resulting WCF program will reside; if this option is omitted, 2314 is assumed.

,VOL=volume serial

Defines the volume serial of the volume on which the resulting WCF program will reside; if this option is omitted, USER08 is assumed.

,REGC=nK (Default is 120K.)

,TIMEC=seconds (Default is 999.)

,PARMC=string

Only one parameter, STMT, is currently significant to the WCFX processor. STMT initiates the printing of WCF generated line numbers on the WCFX listing. These line numbers are particularly useful for debugging WCFX programs. STMT may appear anywhere within the parameter string; all other characters in the string are ignored. If this option is omitted, WCF generated line numbers will not appear on the WCFX listing.

WCFX Return Codes

1000	No errors were detected
1001-1998	1-198 errors detected
1999	A WCFX limitation was encountered that caused an early termination of processing.

WCFX PROGRAM FORMAT

WCFX programs are freeform in the sense that they may begin and end anywhere within a line, and may continue across succeeding lines with blanks and comments imbedded freely. The processor scans only columns 1 through 72 of input lines; all sequences of blanks outside quoted character strings are reduced to a single blank and all comments are deleted from the text as part of the input phase processing. All WCFX statements are terminated by a semicolon (a null statement consisting of only a semicolon is allowed).

Character strings are enclosed in quotation marks ('THIS IS A CHARACTER STRING') and may contain any sequence of characters. A quotation mark within a character string must be specified as two adjacent quotation marks ('DON''T'). Comments are enclosed in exclamation points (!THIS IS A COMMENT!) and may contain any sequence of characters. An exclamation point within a comment must be specified as two adjacent exclamation points (!WHAT!!?!). Comments within character strings are treated as part of the character string, and vice-versa.

The WCFX processor inserts the WCF LET verb in front of all lines containing an assignment operator (=). This can be prevented (and any non-assigment use of = can be indicated) by coding the operator as two adjacent assignment operators (==) that will be translated to a single operator in the resulting WCF program.

All WCF lines generated by the WCFX processor must be less than 124 characters in length. An error message (LINE TRUNCATED) appears in the WCF final program listing following any line that is longer than 123 characters.

Certain WCF constructs may be improperly translated by the WCFX processor because they have the appearance of WCFX statements. There is no general rule for avoiding these situations but the following example may provide some help in working around them.

```
MOVE 5/10 TO END;
```

The "END;" would be processed as a WCFX END statement. This can be prevented by either of the following:

```
MOVE 5/10 TO END,;
```

OR

```
.TEMP='END';
MOVE 5/10 TO %.TEMP;
```

3.1 ELEMENTS OF THE LANGUAGE

LABELS

General Form
label1: statement;
label1: label2: .....labeln: statement;

Labels are any arbitrary sequence of up to 30 characters (except a quotation mark or an exclamation point) followed by a colon. The label begins with the first non-blank character and includes all characters preceeding the colon. Blanks are allowed in a label but their use is discouraged since they can produce some unexpected results. Such results can also occur when labels and variables have identical names, or even when variables have initial portions of their names identical to a label name.

Examples

```
LABEL:;

HERE: THERE: AND_EVERYWHERE:;

SUBRTN: .X=1;

X: Y: DO .I=1 TO 10;
```

ARRAYS

General Form	
.variable[nn]	(maximum 6 character name)
.variable[nn1,nn2]	(maximum 4 character name)
.variable[nn1,nn2,nn3]	(maximum 2 character name)

The WYLBUR Command Facility does not support subscripted variables; however, the appearance of subscripted variables is provided by dynamically mapping them into unique WCF variable names. This is done by evaluating the subscripts and appending them as part of the variable name itself (see examples below). The subscripts can be arbitrary expressions that result in a numerical value in the range 0-99 inclusive.

Caution should be exercised when using subscripted variables since they result in very long (and slow) WCF expressions that can easily exceed the 123 character line length limit imposed on resulting WCF programs. Also, subscripted variables cannot be used inside parenthesized expressions because of the restrictions imposed on the use of the WCF % construct.

Generated WCF Code

.variable[nn] is replaced by

```
.variable%(STRZ(nn,2))
```

.variable[nn1,nn2] is replaced by

```
.variable%(STRZ(nn1,2)||STRZ(nn2,2))
```

.variable[nn1,nn2,nn3] is replaced by

```
.variable%(STRZ(nn1,2)||STRZ(nn2,2)||STRZ(nn3,2))
```

Examples

```
.X[3]
     becomes .X%(STRZ(3,2))
     which maps into .X03


.Y[2,4]
     becomes .Y%(STRZ(2,2)||STRZ(4,2))
     which maps into .Y0204


.Z[5,8,50]
     becomes .Z%(STRZ(5,2)||STRZ(8,2)||STRZ(50,2))
     which maps into .Z050850


.ARRAY[.I]
     becomes .ARRAY%(STRZ(.I,2))
     which maps into .ARRAY10, for example, if .I equals 10
```

SUBROUTINES

General Form

```
CALL label;

label: ...
       .
       .
       .
       RETURN;
```

Subroutine calls are implemented by using a stack to maintain the sequence of return points. Subroutines may be recursive and the maximum depth of nesting is limited only by available storage.

There is no formal means of defining a subroutine in WCFX. The label on a CALL statement may be any valid WCFX statement label. Control passes from the CALL statement to the point defined by the specified label and returns to the statement following the CALL statement upon execution of a RETURN statement.

Generated WCF Code

```
CALL label;

     .@@1=.@@1+1                              line n1
     .@@2%(STRZ(.@@1,2))=n2+20                line n2
     GOTO label                               line n3

RETURN;

     .@@3=.@@2%STRZ(.@@1,2))                  line n1
     .@@1=.@@1-1                              line n2
     GOTO %(.@@3)                             line n3
```

AUTOMATIC VARIABLES

General Form
DECLARE .variable AUTOMATIC;
DECLARE (.variable1 <,.variable2,...>) AUTOMATIC;

All WCF variables are global and static; however, the appearance of a form of non-global storage is provided by dynamically mapping those variables declared AUTOMATIC into unique WCF variable names. These variables appear to the programmer as a stack: each time a CALL is executed a "new" set of variables is provided, and each time a RETURN is executed the previous set of variables is recovered. No assumptions should be made about the initial values of AUTOMATIC variables and they cannot be used inside parenthesized expressions because of the restrictions imposed on the use of the WCF % construct.

Generated WCF Code

Each occurrence of .variable is replaced by

```
.@nnnn%(stringz(.@@1+0,2))
```

where nnnn is a unique number for each AUTOMATIC variable.

Example 1

```
DECLARE .X AUTOMATIC;
.X='FIRST X';
CALL SUBRTN;
TYPE .X;                          (types 'FIRST X')
PAUSE;
```

Example 2

```
SUBRTN:
.X='SECOND X';
RETURN;
```

3.2 LANGUAGE STATEMENTS

IF Statements

General Form

IF expression THEN statement;
 	
IF expression THEN DO;
  statement_list
END;
 	
if_then_part
ELSE statement;
 	
if_then_part
ELSE DO;
  statement_list
END;

The IF statement allows for the conditional execution of either a single statement or a group of statements based on the result of evaluating a specified expression. By including the optional ELSE portion of the statement the test can be used to select which of two statements (or groups of statements) is to be executed.

The present version of the WCFX processor does not permit the subject statement of either the THEN or ELSE constructs to be another IF statement or a DO statement. Nested IF statements can be written by using the THEN DO or ELSE DO forms of the IF. These restrictions will be relaxed in the future.

Generated WCF Code

IF expression THEN statement;

```
     IF expression                          line n1
     ELSE GOTO n4                           line n2
     statement                              line n3
                                            line n4
```

```
IF expression THEN DO;
     statement_list
END;

     IF expression                                    line n1
     ELSE GOTO n3                                     line n2
     statement_list
                                                      line n3

if_then_part
ELSE statement;

     WCF code for if_then_part
     GOTO n3                                          line n1
     statement                                        line n2
                                                      line n3

if_then_part
ELSE DO;
     statement_list
END;

     WCF code for if_then_part
     GOTO n2                                          line n1
     statement_list
                                                      line n2
```

Example_1

```
IF .A EQ .B THEN .A=.A-1;
```

Example_2

```
IF .FALSE THEN DO;
     TYPE 'FALSE';
     .FALSE=1;
END;
```

Example_3

```
IF .X LT 2*.Y THEN PAUSE;
ELSE DO;
     .X=.X+.Y;
     GOTO THERE;
END;
```

UNLESS Statements

General Form

```
UNLESS expression THEN statement;

UNLESS expression THEN DO;
     statement_list
END;


unless_then_part
ELSE statement;

unless_then_part
ELSE DO;
     statement_list
END;
```

The UNLESS construct is the converse of the IF construct (it is equivalent to IF NOT). UNLESS statements are exactly like IF statements in syntactic form and are subject to the same restrictions.

Generated WCF Code

```
UNLESS expression THEN statement;

     IF expression                                  line n1
     THEN GOTO n4                                   line n2
     statement                                      line n3
                                                    line n4

UNLESS expression THEN DO;
     statement_list
END;

     IF expression                                  line n1
     THEN GOTO n3                                   line n2
     statement_list
                                                    line n3
```

```
unless_then_part
ELSE statement;

     WCF code for unless_then_part
     GOTO n3                                          line n1
     statement                                        line n2


unless_then_part
ELSE DO;
     statement_list
END;

     WCF code for unless_then_part
     GOTO n2                                          line n1
     statement_list
                                                      line n2
```

Example_1

```
UNLESS .A EQ .B THEN .A=.A+1;
```

Example_2

```
UNLESS .FALSE THEN DO;
     TYPE 'TRUE';
     .FALSE=0;
END;
```

Example_3

```
UNLESS 2*.X LT .Y THEN PAUSE;
ELSE GOTO SOME_MORE;
```

DO Statements

General Form

```
DO WHILE expression;
     statement_list
END;

DO UNTIL expression;
     statement_list
END;

DO .variable=expression1 TO expression2;
     statement_list
END;

DO .variable=expression1 TO expression2 BY expression3;
     statement_list
END;
```

The DO WHILE construct indicates that the associated statement list is to be executed repetitively as long as the specified expression is true. The DO UNTIL construct is the converse of the DO WHILE and indicates that the associated statement list is to be executed repetitively as long as the specified expression is false. The test of the expression is made at the beginning of each repetition of the loop for WHILE statements and and at the end of each repetition of the loop for UNTIL statements (UNTIL loops are always executed at least once).

The iterative forms of the DO statement indicate that the associated statement list is to be executed repetitively with the control variable set initially to the value of expression1, then to expression1+expression3, then to expression1+2*expression3, ..., until its value is greater than (or less than) the value of expression2. If expression3 is omitted an implied value of 1 is used. The direction in which the value of the control variable is changed and the termination condition for the loop are determined from the sign of expression3.

There are no restrictions on branching into or out of any of the DO statements and all variables associated with a DO statement may be changed freely.

Generated WCF Code

```
DO WHILE expression;
     statement_list
END;

     IF expression                                    line n1
     ELSE GOTO n4                                     line n2
     statement_list
     GOTO n1                                          line n3
                                                      line n4

DO UNTIL expression;
     statement_list
END;

     GOTO n4                                          line n1
     IF expression                                    line n2
     THEN GOTO n6                                     line n3
                                                      line n4
     statement_list
     GOTO n2                                          line n5
                                                      line n6

DO .variable=expression1 TO expresssion2;
     statement_list
END;

     .variable=expression1                            line n1
     GOTO n4                                          line n2
     .variable=.variable+(1)                          line n3
     IF (1)*(.variable-(expression2)) GT 0            line n4
     THEN GOTO n7                                     line n5
     statement_list
     GOTO n3                                          line n6
                                                      line n7

DO .variable=expression1 TO expression2 BY expression3;
     statement_list
END;

     .variable=expression1                                line n1
     GOTO n4                                              line n2
     .variable=.variable+(expression3)                    line n3
     IF (expression3)*(.variable-(expression2)) GT 0      line n4
     THEN GOTO n7                                         line n5
     statement_list
     GOTO n3                                              line n6
                                                          line n7
```

Example 1

```
DO WHILE .X LT 10;
     TYPE .X;
     .X=.X+1;
END;
```

Example 2

```
DO UNTIL (.Y LT 10) OR (.A NE .B);
     .A=2*.Y;
     .Y=.Y-1;
END;
```

Example 3

```
DO .I=100 TO -10 BY -3;
     TYPE .I;
END;
```

NEXT and EXIT Statements

General Form

```
label: do_statement
          .
          .
          .
          NEXT label;
          .
          .
          .
          EXIT label;
          .
          .
          .
       END;
```

NEXT and EXIT statements can appear only as part of the statement list of labeled DO statements. Furthermore, the label specified on these statements must be the first label on the DO statement if more than one label is used.

The NEXT statement generates a branch to the closing END statement of the labeled DO statement to which it refers and effectively causes the next iteration of the loop to occur. The EXIT statement generates a branch to the statement immediately following the closing END statement of the labeled DO statement to which it refers and effectively causes the loop to be exited. NEXT and EXIT statements are particularly useful for defining the flow of control in the nested DO statements.

Generated WCF Code

NEXT label;

```
     GOTO n2                                    line n1
     .
     .
     .
     GOTO for do_statement (see above)          line n2
```

```
EXIT label;

     GOTO n3                                        line n1
     .
     .
     .
     GOTO for do_statement (see above)              line n2
                                                    line n3
```

Example_1

```
LOOP: DO WHILE .TRUE;
     .
     .
     .
     NEXT LOOP;
     .
     .
     .
     EXIT LOOP;
     .
     .
     .
    END;
```

Example_2

```
HERE: THERE: DO UNTIL .FALSE;
ANOTHER: DO .I=1 TO 10;
     .
     .
     .
     EXIT HERE;
     .
     .
    END;
    END;
```

3.3 WCFX ERROR MESSAGES

Since the WCFX processor is a macro-processor and not a compiler, very little syntax checking is done. Usually, syntax errors result in patterns that do not match the WCFX macros properly and which are, therefore, passed on as part of the WCF program. These errors are easily detected by scanning the resulting WCF program for untranslated constructs.

There are three program related error messages produced by the WCFX processor:

INVALID BLOCK STRUCTURE - ILLEGAL "END/ELSE" STATEMENT

This error usually results from an END or ELSE statement that has no corresponding initial part (DO statement or IF for END, IF or UNLESS for ELSE). This message may be generated multiple times for a single END or ELSE statement.

INVALID BLOCK STRUCTURE - INSUFFICIENT "END" STATEMENTS

This error is the converse of the previous one. An end-of-file occurred before the logical end of the program leaving at least one unclosed block.

LINE TRUNCATED

This message follows those WCF generated lines that have been truncated because their length was greater than 123 characters.

There are two error messages related to built-in limitations of the WCFX processor itself.

STACK OVERFLOW

This error results from exceeding the processor's maximum IF/UNLESS and/or DO statement nesting capability. The maximum nesting level allowed is printed on the first page of every WCFX listing (currently 50).

MACRO/LABEL OVERFLOW

This error results when the number of labels required by a program exceeds the number that can be contained in the processor tables. This number can be automatically increased by increasing the region allocated to the processor. The maximum number of labels allowed is printed on the first page of every WCFX listing (PASS 2 MACROS ALLOWED), and the number of labels defined by a program is printed on the last page of every WCFX listing (PASS 2 MACROX DEFINED).

WCFX PROGRAM EXAMPLE -- WCFX Source

The following example is intended as a vehicle for displaying some of the features of the WCFX language. It is not meant to be a good example of how to compute factorials or of how WCF should be used.

```
     DECLARE .X AUTOMATIC;
AGAIN:
     REQUEST NUM .N PROMPT'FACTORIAL FOR N= ';
     IF (.N LT 0) OR (TRU(.N) NEQ .N) THEN TYPE 'DUMMY';
     ELSE DO;
          IF .N EQ 0 THEN PAUSE 'GOODBYE';
          ELSE DO;
               .Z=.N;
               CALL FACT;
               TYPE STR(.N) || ' FACTORIAL = ' || STR(.Y);
          END;
     END;
     GOTO AGAIN;


FACT:
     IF .Z EQ 1 THEN DO;
        .Y=1;
         RETURN;
     END;
     ELSE DO;
         .X=.Z;
         .Z=.Z-1;
         CALL FACT;
         .Y=.X*.Y;
         RETURN;
     END;
```

Program Output -- Generated WCF Program

```
0010.000 REQUEST NUM .N PROMPT'FACTORIAL FOR N= '
0020.000 IF .N LT 0 OR TRU(.N) NEQ .N
0030.000 ELSE GOTO 0060
0040.000 TYPE 'DUMMY'
0050.000 GOTO 0150
0060.000 IF .N EQ 0
0070.000 ELSE GOTO 0100
0080.000 PAUSE 'GOODBYE'
0090.000 GOTO 0150
0100.000 LET .Z=.N
0110.000 LET .@@1=.@@1+1
0120.000 LET .@@2%(STRZ(.@@1,2))= 0120+20
0130.000 GOTO 0160
0140.000 TYPE STR(.N) || ' FACTORIAL = ' || STR(.Y)
0150.000 GOTO 0010
0160.000 IF .Z EQ 1
0170.000 ELSE GOTO 0230
0180.000 LET .Y=1
0190.000 LET .@@3=.@@2%(STRZ(.@@1,2))
0200.000 LET .@@1=.@@1-1
0210.000 GOTO %(.@@3)
0220.000 GOTO 0320
0230.000 LET .@0010%(STRZ(.@@1+0,2))=.Z
0240.000 LET .Z=.Z-1
0250.000 LET .@@1=.@@1+1
0260.000 LET .@@2%(STRZ(.@@1,2))= 0260+20
0270.000 GOTO 0160
0280.000 LET .Y=.@0010%(STRZ(.@@1+0,2))*.Y
0290.000 LET .@@3=.@@2%(STRZ(.@@1,2))
0300.000 LET .@@1=.@@1-1
0310.000 GOTO %(.@@3)
```